An Introduction to UVM

Ryan Billmeyer

October 14, 2022

Abstract

The notes are adapted from The UVM Primer by Ray Salemi.

1 Introduction

These notes cover a working introduction to the Universal Verification Methodology (UVM). The goal is to be familiar enough with the UVM to interface with and use it. Sections of code are stylized as

```
module top;
    tinyALU(.*);
endmodule
```

2 The Conventional Testbench

UVM fundamentally builds off the SystemVerilog testbench architecture, so we will start there. The goal of our testbench is to implement a "coverage-first" methodology by

- Exhaustively testing the DUT functionality
- Exercising all lines of RTL
- Exercising all datapaths through the DUT

In addition, our testbench should be self-checking so we don't have to sift through the results.

2.1 TinyALU

The design that will be referenced throughout these notes is called TinyALU. As the name implies, this is a tiny arithmetic logic unit. It takes in two 16-bit operands, a 3 bit op-code, and a start bit. TinyALU outputs a 16-bit result and a done signal. As a verbal timing diagram: The start signal is asserted high on the posedge of the clk, the operands and op-code are read in, the operation is performed in N cycles, the result is calculated and the done signal is raised, then the start signal is lowered. The design is reset through a synchronous active low reset line. It supports 5 operations: NOP (3'b000), ADD (3'b001), AND (3'b010), XOR (3'b011), MULT (3'b100).

2.2 TinyALU Testbench

The components of the simple SV testbench include a top level declaration, a coverage block with covergroups, a monitor, a driver, a generator and a scoreboard. The coverage block measures what functionality we have tested. The scoreboard achieves self-checking. The generator creates constrained random stimulus. This architecture is discussed more in-depth in SystemVerilog for Verification by Chris Spear.

3 Interfaces and Bus Functional Models

The ultimate targets for any verification environment are modularity and abstraction. An important concept towards these goals is the SystemVerilog interface. Interfaces encapsulate the port signals of a module or object, greatly simplifying how we connect to them. Interfaces can then be used to encapsulate bus protocols into simple routines through Bus Functional Models (BFM).

3.1 Bus Functional Model (BFM)

A BFM is an interface. For the TinyALU, our BFM will contain all the signals needed, provide a clock, a reset task, and a send op task.

```
interface tinyalu_bfm ();
    byte unsigned A;
    byte unsigned B;
    logic clk;
    logic reset_n;
    logic [2:0] op;
    logic start;
    logic done;
    logic [15:0] result;

    modport dest(
        input A,
        input B,
        input clk,
        input reset_n,
        input op,
        input start,
        output done,
        output result
    );

    modport src(
        output A,
        output B,
        input clk,
        output reset_n,
        output op,
        output start,
        input done,
        input result
    );

    task gen_clk();
        clk = 0;
        forever begin
            #1000
            clk = ~clk;
        end
    endtask : gen_clk

    task reset_alu();
        reset_n = 1'b0;
        @(negedge clk);
        @(negedge clk);
        reset_n = 1'b1;
        start   = 1'b0;
    endtask : reset_alu

    //Defines protocol to interact with DUT given op
    task send_op(input byte iA, iB, input logic [2:0] iop,
    output var logic [15:0] result);
        op = iop;
        if(iop == rst_op) begin
            @(posedge clk);
            reset_alu();
        end else begin
            @(negedge clk);
            A = iA;
            B = iB;
            start = 1'b1;
            if (iop == no_op) begin
                @(posedge clk);
                #1;
                start = 1'b0;
            end else begin
                do
                    @(negedge clk);
                while(done == 0);
                start = 1'b0;
            end
        end
    endtask : send_op

endinterface : tinyalu_bfm
```

Placing our bus protocol within our BFM allows us to instantiate this interface anywhere we need to drive our TinyALU from a testbench. In addition, if our protocol is updated, we only need to update the send op task in our BFM to update all testbenches.

3.2 Modularizing the Testbench with BFMs

The definition of our BFM allows us to connect the pieces of our verification environment using an interface instead of a port list.

```
module top;
    tiny_alu_bfm bfm();
    random_tester random_tester_i (bfm.src);
    coverage coverage_i (bfm.dest);
    scoreboard scoreboard_i (bfm.dest);

    tiny_alu DUT (bfm.dest);
endmodule
```

This benefit extends into the verification objects themselves too. They can now utilize the tasks defined in the BFM.

```
module scoreboard(tiny_alu_bfm bfm);
    always @(posedge bfm.done) begin
        logic [15:0] pred_result;
        case (bfm.op)
            add_op: pred_result = bfm.A + bfm.B;
            ...
        endcase
        if((bfm.op != no_op) && (bfm.op_set != rst_op))
            if(pred_result != bfm.result)
                //report failure
    end
endmodule : scoreboard
```

In the same sense the tester can also be simplified along the lines:

```
module tester (tiny_alu_bfm bfm);
    initial begin
        bfm.reset_alu();
        repeat(1000) begin : random_loop
            op = get_op();
            iB = get_data();
            iA = get_data();
            bfm.send_op(iA,iB, op, result);
        end : random_loop
        $finish
    end
endmodule
```

4 Brief Notes on OOP in SystemVerilog

These topics are fairly basic. I'm highlighting some areas that may be tricky.

4.1 Constructor Inheritance

An extended class must have a constructor defined if an argument is passed into the constructor at instantiation. This means that even if we just intend to call the parent constructor, we still need to define a constructor in the extended class if the parent constructor requires an argument. This is the common case. As a side note, the compiler will be able to tell what constructor you are calling based on the reference type of the object on the LHS.

4.2 Memory allocation for classes

A class object must be instantiated. First, we declare the handle for the class object. Then, we instantiate the object and store the instantiation in the handle.

```
    class class_i; //Creating handle
    class_i = new(); //instantiate the object and assign to handle
```

Memory for the handle is allocated once it is created. Memory for the non-static class object is allocated at instantiation. Passing the object between environment components is the same as passing the memory space between threads, hence why mailboxes are used for synchronization.

4.3 Virtual and Abstract

OOP allows us to store an extended class object in a base class object. However, we run into issues with what methods we are referring to.

Virtual methods solve these issues and are common to all OOP languages. My favorite explanation of a virtual method is a method where the definition depends on the runtime type of the underlying object. On the other hand, a non-virtual method is a method where the definition depends on the reference type of the object at the point of invocation. This means that if an extended class object is instantiated and assigned to a base class object:

- All invoked virtual methods will reference the method definition that is associated with the class type we instantiated and not the one we assigned the object to.

- All invoked non-virtual methods will reference the method definition that is associated with the class type the object is assigned to at the point of invocation.

In essence, a virtual method says that the definition lies somewhere else. An overridden virtual method will inherit the virtual property, so the virtual keyword is not necessary in the extended class.

```
class base_class;
    int a;
    function new(int a);
        this.a = a;
    endfunction : new

    virtual function void method1();
    endfunction : method1
endclass : base_class

class ext_class extends base_class;
    int a;
    function new(int a);
        super.new(a);
    endfunction : new

    //Inherits virtual keyword
    function void method1();
        //override base definition
    endfunction : method1
endclass : base_class

module tb;
    base_class base_class_h;
    ext_class ext_class_h;
    ext_class_h = new(); //instantiate extended class
    base_class_h = ext_class_h; //assign extended class object to base class handle

    If method1 is virtual:
        base_class_h.method1(); //this will invoke method1 definition in ext_class_h
    If method1 is not virtual:
        base_class_h.method1(); //this will invoke method1 definition in base_class_h

endmodule
```

We can take this one step further with the notion of abstract classes and pure virtual methods. These two concepts can be enforced to remove the ambiguity of what method definition we are referring to. Abstract classes can only be base classes; they cannot be instantiated. The methods of an abstract class can be defined as pure virtual methods. Pure virtual methods only specific the declaration of the method. The method definition **must** be provided by an extended classes. Abstract classes may also contain virtual or non-virtual methods.

```
//Define an abstract class
virtual class abs_class;
    int a;

    function new (int a);
        this.a = a;
    endfunction

    //Only provides declaration of method
    pure virtual function void method1();
```

```
endclass : abs_class

class ext_class extends abs_class;
    int a;

    function new (int a);
        super.new(a);
    endfunction

    //Override method1 definition
    function void method1();
    endfunction
endclass : ext_class
```

4.4 Static

The static keyword can be applied to variables and methods. It is an intelligent approach to the tricky subject of global resources.

A static data member of a class can be thought of as a global variable that is shared between instances of that class. No matter how many times you instantiate the class, you will always be pointing to the same memory location. As such, the memory for a static data member of a class is allocated at the time the class handle is created. Additionally, the static member can be accessed without instantiating the class. For example, say we want a queue of objects to be visible from anywhere in the testbench.

```
class transaction_queue;
    static transaction q[$]; \\queue in SV
endclass

module top;
    transaction transaction_h;
    transaction_h = new(...);
    transaction_queue::q.push_back(transaction_h);
endmodule
```

We extend on this with static methods that can restrict the operations that can be performed on our static data members. The protected keyword means this data member can only be accessed through the class methods, not limited to static cases.

```
class transaction_queue;
    protected static transaction q[$]; \\queue in SV
    static function void push (transaction tran);
        this.q.push_back(tran);
    endfunction
endclass

module top;
    transaction transaction_h;
    transaction_h = new(...);
    transaction_queue::push(transaction_h);
endmodule
```

4.5 Parameterized Class Definitions

The ability to parameterize classes is a natural extension of the HDL ability to parameterize IP blocks. In addition to data variable parameters, we can create type parameters for a class. This allows us to specify the reference type we are operating on at invocation. Take the above static method example. We can enhance our transaction queue class to work with any object type by adding a type parameter.

```
class base_queue #(type T);
    protected static T q[$]; \\queue of type T objects in SV
    static function void push (T obj);
        this.q.push_back(obj);
    endfunction
endclass : base_queue

module top;
    transaction transaction_h;
    driver      driver_h;
    transaction_h = new(...);
    driver_h = new(...);
    base_queue #(transaction)::push(transaction_h); \\specify ref type of trans
    base_queue #(driver)::push(driver_h);
endmodule
```

It is important to note that specifying a different parameter value will create two copies of the base queue: one for transaction type objects and one for driver type objects. By changing the parameter, we are creating a completely new class with that parameter. The classes do not share namespaces.

While the static keyword is useful, it is not always appropriate. Thus, we often use parameterized classes to declare dynamic variables within a defined scope. Let's show this by translating the above example into non-static terms.

```
class base_queue #(type T);
    protected T q[$]; \\queue of type T objects in SV
    function void push (T obj);
        this.q.push_back(obj);
    endfunction
endclass : base_queue

module top;
    transaction transaction_h;
    driver      driver_h;
    base_queue #(transaction) trans_q;
    base_queue #(driver) dvr_q;
    initial begin
        dvr_q = new();
        trans_q = new();
        //invoke methods and stuff
    end
endmodule
```

The instantiated queues now only exist within the scope of module top, not the entire testbench. The memory they occupy will also be de-allocated upon completion of the task.

5 The Factory Pattern

Up until this point, we have been hard-coding the instantiation of our classes using new(). This is bad. We very quickly run into flexibility issues with this approach. What if we want to change the transaction and driver objects to bicycle and car objects? We would have to edit the source code and recompile, wasting away the best years of our life. We can avoid this issue by using the factory pattern. As the name implies, we want to mimic a factory that produces objects. The factory takes in raw materials (constructor arguments) and uses a blueprint (constructor) to output a product (a class object of the desired type). The benefit of this approach is that we can swap the blueprint for a different blueprint (constructor for a different class extended from the same base class) at any time and get our desired output. Swapping the blueprints is handled automatically by our OOP approach; all we need to do is alter the constructor argument. That's a very power technique; the ability to generate a specific object at run time by changing a parameter is not to be understated.

5.1 Creating a Factory

The factory is implemented, unsurprisingly, as a class. There is a static method that is passed the arguments we are interested in and the type of object we wish to create. Given these, the method returns the class object.

```
class car;
    int length, width;
    function new (int length, width);
        this.length = length;
        this.width  = width;
    endfunction

    pure virtual function void drive();

endclass : car

class sedan extends car;
    int avg_mpg;
    function new(int length, width, mpg = 0);
        super.new(length,width)
        this.avg_mpg = mpg;
    endfunction

    function void drive();
        //automatic
    endfunction
endclass

class suv extends car;
    int avg_mpg;
    function new(int length, width, mpg = 0);
        super.new(length,width)
        this.avg_mpg = mpg;
    endfunction

    function void drive();
        //manual
    endfunction
endclass

class car_factory;
    static function car make_car(string model, int length, width);
        suv suv_h;
        sedan sedan_h;
        case (model)
            "sedan" : begin
                sedan_h = new(length, width)
                return sedan_h;
            end
            "suv" : begin
                suv_h = new(length, width)
                return suv_h;
            end
            default begin
                $fatal();
            end
        endcase
    endfunction
endclass
```

Thanks to polymorphism, we can return the sedan object as a car type, allowing us to dynamically create different kinds of cars.

5.2 Using the Factory

Since the make car function is static, it can be invoked anywhere within the testbench. The use of a pure virtual drive method allows the invocation of the drive method to correspond to the correct reference type.

There may arise an issue with returning these derived classes as a base class. Specifically, if we tried to access the avg mpg member of the sedan class, we would get an error using the above method because avg mpg is not a member of the car base class. To remedy this, we need to cast the returned car as a sedan.

```
car car_h;
car_h = car_factory::make_car("sedan",1,2);
car_h.drive(); //invoke sedan drive function
cast_ok = $cast(sedan_h, car_h); //cast car as sedan
```

The cast function requires that the class being casted to is a child class of the class we are casting from. We can condense all of this into a single line.

```
if(!$cast(sedan_h, car_factory::make_car("sedan",1,2)))
    $fatal();
base_queue #(sedan)::push(sedan_h);
```

The ability of the factory will be useful to dynamically create our testbench components.

6 An Object-Oriented Testbench

We now wish to transition our module-based testbench towards an object-oriented one. The first pass at an object-oriented testbench for the TinyALU will consist of one module and four classes:

- Top - Top-level module that instantiates the testbench class
- Testbench - Top-level class
- Tester - Drives the stimulus
- Scoreboard - Checks observed results against predicted results
- Coverage - Captures functional coverage information

6.1 Top-level module

The top-level module is mainly responsible for doing three things:

- Importing class definitions
- Instantiating the DUT and BFM and declaring the testbench handle
- Instantiating and executing the testbench class

These objectives are fairly self-explanatory and best showcased via writing out the top module.

```
module top;
    //1. Import class definitions + macros
    import tinyalu_pkg::*;
    'include "tinyalu_macros.svh"

    //2. Instantiate the DUT, BFM
    tinyalu DUT (bfm.dest);
    tinyalu_bfm bfm();

    //2. Declare testbench handle
    testbench testbench_h;

    //3. Instantiate and execute testbench
    initial begin
        testbench_h = new(bfm);
        testbench_h.execute();
    end
endmodule
```

6.2 Testbench Class

The testbench class is the top level of the testbench hierarchy. The responsibility of the testbench class is to instantiate, connect, and evoke methods from the other 3 classes in our verification environment. The UVM will have written this already. Once again, this is clearest by giving code first.

```
class testbench;
    virtual tinyalu_bfm bfm;

    tester tester_h;
    scoreboard scoreboard_h;
    coverage coverage_h;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        //instantiate and connect helper classes
        tester_h = new(this.bfm);
        scoreboard_h = new(this.bfm);
        coverage_h = new(this.bfm);

        //execute classes
        fork
            tester_h.execute();
            scoreboard_h.execute();
            coverage_h.execute();
        join_none
    endtask : execute
endclass : testbench
```

There are two new things in this piece of code that are worth noting.

The first is the use of the keyword **virtual** on the bus functional model interface. This keyword allows us to pass the interface into a class. The keyword virtual is saying that the we will be passing the variable bfm a handle to an interface at some point, but not at compile time (modules are not dynamic objects, which explains why we don't use virtual interfaces with them). We've essentially told the compiler that we got this part, so now we need to actually keep our promise. The approach we take in the above example is simply passing the handle through the constructor, new(bfm).

The second new thing is the fork/join_none. This is how SystemVerilog implements multi-threading. In this case, each line starts a new execution thread and the code will not wait for any of the threads to finish before continuing execution of the parent thread. Other options include fork/join_all, which waits for all threads to complete, and fork/join_any, which waits for any thread to complete.

6.3 Tester Class

This class is called the tester class in the UVM primer. It is also commonly split into two classes called the driver (drives pin stimulus) and the generator (generates the random stimulus).

```
class tester;
    virtual tinyalu_bfm bfm;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        repeat(N) begin:
            //generate stimulus
            //drive stimulus
        end
        $finish()
    endtask : execute
endclass : tester
```

6.4 Scoreboard Class

The scoreboard class will observe the result from the DUT, calculate its own result, and compare the two. The scoreboard in the UVM primer combines the function of what is commonly called a monitor (translates DUT output into result) and a scoreboard (predicts result and self-checks).

```
class scoreboard;
    virtual tinyalu_bfm bfm;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        forever begin : self_checker
            @(posedge bfm.done)
            //record output
            //predict output
            //self check
        end : self_checker
    endtask : execute
endclass : scoreboard
```

6.5 Coverage Class

The coverage class is also usually incorporated into a scoreboard.

```
class coverage;
    virtual tinyalu_bfm bfm;
    //define covergroup

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
        covergroup_1 = new();
        covergroup_2 = new();
    endfunction : new
    task execute();
        //update covergroups
    endtask : execute
endclass : coverage
```

7 UVM Tests

Now the fun begins. We can officially start diving into the UVM. The main driving force behind the UVM is to systematically enable a dynamic testbench. This is a fancy way of saying we want to run many tests with one compile. The object-oriented testbench that was covered in the last section is still hard-coded in the eyes of a UVM purist. We will now dive into how the UVM constructs a dynamic testbench.

7.1 Creating Tests with the Factory Pattern

The UVM is equipped with its own factory that allows us to build almost anything, which means we can create dynamic testbenches. To start, we want to use the factory to launch different tests.

```
vsim testbench -coverage +UVM_TESTNAME=add_test
vsim testbench -coverage +UVM_TESTNAME=random_test
```

In the above, we are compiling the testbench once then supplying two different test names at run time to execute each test. We define these test names as classes then call the UVM factory with the test names using UVM_TESTNAME. The factory will create instances of the test classes then launch the simulation.

7.2 Launching Simulations with UVM

We will see a trend with the UVM going forward. That trend is automating the creation of the object-oriented testbench. We start with seeing how the UVM automatically creates the equivalent of the top-level testbench class from our OO testbench of last section.

Instead of instantiating our testbench class then calling the execute task, we now let the UVM do the work:

```
module top;
    //These come from the UVM
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    //These are supplied by us
    import tinyalu_pkg::*;
    `include "tinyalu_macros.svh"

    tinyalu_bfm bfm();
    tinyalu DUT (bfm.dest);

    initial begin
        uvm_config_db #(virtual interface tinyalu_bfm)::set(null, "*", "bfm", bfm);
        run_test();
    end
endmodule : top
```

Alright, we still have to the same two things to let UVM do the work: store a handle to the BFM and call the run test method.

Remember how I said there were multiple approaches to passing an interface handle in the last section? The above code features a different approach. Here, we can't pass our interface handle via a constructor because the UVM requires specific arguments for its own constructor. Instead of exposing the constructor, we use the uvm_config_db class. The set method of this class allows us to make the

BFM available across the entire testbench. The arguments null and "*" make the data available across the entire testbench. The third argument supplies the string that names the data we're storing in the database. The fourth argument is the value being stored, the handle to the bfm in our case.

The invocation of the run test method will read the +UVM_TESTNAME parameter and use the UVM factory to instantiate a object of the test class name, whose definition is provided in our own package.

7.3 Defining and Registering a UVM Test

We need to define the test classes such that they are compatible with what the UVM expects to see.

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test); //register test name in UVM factory

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        //Call parent constructor
        super.new(name, parent);
        //Grab bfm handle from database
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
    endfunction : new
endclass
```

A couple things to note. First, our random test class extends the uvm_test class which extends the uvm_component class. In order to satisfy our base class constructor, we need to pass it a name of type string and a parent of type uvm_component (these are explained later). Second, we invoke the get method from the uvm_config_db class to retrieve our class handle and store it in the bfm variable. Finally, the `uvm_component_utils macro registers the random test class with the factory and allows us to generate objects of this type.

7.4 The Run Phase Method

The test object created from the UVM factory will inherit a run_phase method. After being created, the UVM will invoke this run phase method to execute our test. As a result, we need to override the run phase method in our test class. Some strict rules on this: the task must be called run_phase and it must have a single arugment of type uvm_phase called phase.

```
task run_phase(uvm_phase phase);
    random_tester random_tester_h;
    coverage coverage_h;
    scoreboard scoreboard_h;

    phase.raise_objection(this);

    random_tester_h = new(bfm);
    coverage_h = new(bfm);
    scoreboard_h = new(bfm);

    fork
        coverage_h.execute();
        scoreboard_h.execute();
    join_none

    random_tester_h.execute();
    phase.drop_objection(this);
endtask : run_phase
```

This method will be invoked by the UVM after our random test class object is created.

The phase object invokes a method called raise objection to indicate that we (this in OOP speak) are currently running our test and would prefer if the over-arching simulation was not stopped. By invoking the drop objection method, we are saying that (this) thread no longer objects to the simulation stopping. After all objections are resolved, we can safely stop the simulation.

7.5 Extending to Several Test Classes

Any additional test class we want to add follow the same formula:

```
class nop_test extends uvm_test;
    `uvm_component_utils(nop_test); //register test name in UVM factory

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        //Call parent constructor
        super.new(name, parent);
        //Grab bfm handle from database
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
    endfunction : new

    task run_phase(uvm_phase phase);
        nop_tester nop_tester_h; //Create handle for tester object
        //rest of test
    endtask
endclass
```

8 UVM Components

The UVM component class is the building block of the structure of a UVM testbench. As a result, we are often required to define and instantiate classes that extend the UVM component class, like we did for the test classes in the last section. There is a four step process to make any generic class into a UVM component.

- Extend the uvm_component class or child class to define your component
- Use the `uvm_component_utils() macro to register the class in the UVM factory
- Provide at least the minimum uvm_component constructor arguments
- Override the UVM phase methods as necessary

We illustrate this process by re-factoring the tester, coverage, and scoreboard classes from the last section as UVM components.

```
class scoreboard extends uvm_component; //Req. 1
    `uvm_component_utils(scoreboard); //Req. 2

    virtual interface tinyalu_bfm bfm;

    function new(string name, uvm_component parent);
        super.new(name, parent); //Req. 3
    endfunction : new
endclass : scoreboard
```

You will notice that requirement 4 is missing from the above code. Every UVM component will inherit many phases from the base class. The UVM will step through these phases over the course of the simulation lifetime. You can customize your component by overriding a specific phase in the derived class. All phase methods take one argument: phase of type uvm_phase. The phase methods do not need to be overridden. When you do override, make sure to call the parent phase method to ensure you're not creating dead code. A couple phase method examples include:

- build_phase - UVM builds testbench hierarchy from top down. UVM components are instantiated here.
- connect_phase - UVM connects components together.
- end_of_elaboration_phase - UVM calls this after all components are connected.
- run_phase - UVM calls this task in its own thread. All run_phase methods will execute concurrently.
- report_phase - Runs after last objection is dropped and shows stats.

We first override the build phase in the scoreboard class:

```
function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
function : build_phase
```

By retrieving the bfm from the database, we no longer have to pass it into the class in the top-level.

Second, we override the run phase method. Note that run phase is the only phase method that is a task, the rest are functions. Run phase must be a task becomes it can consume simulation time if waiting for a clock edge.

```
task run_phase(uvm_phase phase);
    shortint pred_result;
    forever begin : check_loop
        @(posedge bfm.done)
            case(bfm.op)
        //calculate predicted
        //Compare to actual
    end : check_loop
endtask : run_phase
```

8.1 Building a Testbench

The test classes we created in the last section can actually be simplified using the transition to UVCs and the newly introduced phase methods:

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test); //register test name in UVM factory

    base_tester tester_h;
    coverage coverage_h;
    scoreboard scoreboard_h;

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function build_phase (uvm_phase phase);
        //use uvm_component constructors
        tester_h = new("tester_h", this);
        coverage_h = new("coverage_h", this);
        scoreboard_h = new("scoreboard_h", this);
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
    endfunction
endclass

//Inherits build phase and bfm
class add_test extends random_test;
    `uvm_component_utils(add_test);
    add_tester tester_h; //override tester class

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass
```

9 UVM Environments

Using the factory to create a top-level UVM test then having the test instantiating UVM components can lead to issues with re-usability. We want to target a solution that uses adaptable programming, where the testbench grows stronger every time we add a feature to it. This is in contrast to intractable programming, where expanding the testbench makes using it more complicated.

Adaptable coding has three guidelines:

- Create classes that do one thing very well and connect them to build a solution
- Avoid hardcoding anything if possible
- Program around interfaces, not making assumptions about the underlying implementation

Currently, our test class objects violate the first guideline because they both create the testbench structure by instantiating components **and** modify the testbench behavior by declaring different types of tester classes. We wish to move the testbench structure portion to the UVM environment class, uvm_env.

Before we do this, let's look at an example of an adaptable testbench. I'm going to attempt to switch to UML-like diagrams here. First, we see the intractable solution:

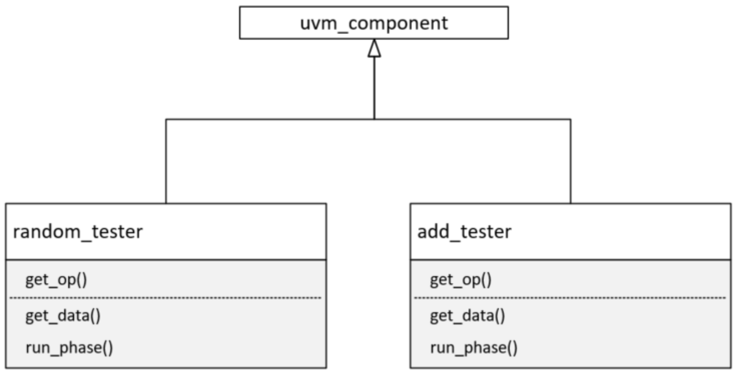

Figure 1: An intractable testbench

There are several issues with the testbench in Figure 1. A different copy of the run phase method exists in each class. Meaning that if we wanted to alter this method, we would have to update it individually in all classes that use it. This lends to very poor scaling abilities.

Now, let's come up with an adaptable version of this testbench. The first thing we notice is that the run phase method is copied in each class. This means we can move it up a level of the class hierarchy and let the child classes inherit it instead, giving us one central definition of the run phase method. The same can be said about get op and get data methods. In both classes, get data returns constrained random data to send over. In contrast, the get op method will return a random operation in the random tester and an add operation in the add tester. This indicates that we should keep the two classes separate. However, we can think of the add operation as a subset of all possible operations from the random tester. As a result, it is most intuitive to make the add tester a child class of the random tester that will inherit the get data method and override the get op method. This is shown in UML diagram form below:

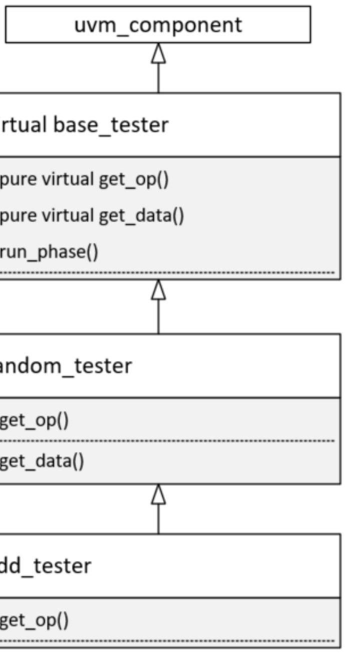

Figure 2: An adaptable testbench

The adaptability of the testbench in Figure 2 can best be shown by thinking about the effort that would be required to add a new test class, call it xor tester:

```
class xor_tester extends random_tester;
    `uvm_component_utils(xor_tester);

    function logic [2:0] get_op();
        return xor_op;
    endfunction : get_op

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass : xor_tester
```

This test class will exercise the xor functionality of the ALU. It is a very simple class definition because it has inherited the run phase method and the get data method.

9.1 Separating Structure from Stimulus

Think back to the above random test and add test classes. In the random test class, we defined the structure of the testbench as a random tester, coverage, and scoreboard. Every test class derived from the random test class must now use this specific testbench makeup. We can avoid this limitation by moving the testbench structure out of the random test class and into the UVM environment class. The uvm_env class normally only contains the build phase and connect phase methods.

The uvm_env class holds the structure of the testbench, then we use the test to specify what objects will fill in the structure. Think of it like a block game that a baby plays. The structure of the testbench determines what the shape of the holes are. The test class will specify what color block goes in each hole. So it is up to the test class to decide if a red square or a blue square is used, but the presence of a square hole is decided by the environment class. Here is the code to define the env class:

```
class env extends uvm_env;
    `uvm_component_utils(env);

    //Create handles for the three UVCs
    base_tester tester_h;
    coverage coverage_h;
    scoreboard scoreboard_h;

    //Instantiate the UVCs
    function void build_phase (uvm_phase phase);
        tester_h = build_tester::type_id::create("tester_h",this);
        coverage_h = coverage::type_id::create("coverage_h",this);
        scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
    endfunction : build_phase

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass : env
```

9.2 Creating UVCs with the UVM Factory

There are some new lines in the above code that seem complex. These are necessary to utilize the UVM factory to create a UVC. The UVM factory is better at making UVCs than our first attempt at a factory was. It is able to return the correct type of object, so we no longer have to cast the object back to the desired type. All we need to do to create a UVC with the UVM factory is the following:

```
<UVC class variable> = <UVC class name>::type_id::create("<UVC class variable>",this);
```

Invoking this static method will spit out compile time errors if you forget to register the UVC, define the UVC, or misspell the UVC.

One lingering sticking point from the above env class is the instantiation of the tester. A sharp eye will notice that it seems like we are instantiating the build tester class, however; this class is an abstract class, which can't be instantiated. The UVM factory allows this because it is making the assumption that we will override the base tester class in the factory with a child class of the base tester class before build phase is called. This is called a factory override.

A factory override takes the form:

```
<base_UVC_name>::type_id::set_type_override(<child_UVC_name>::get_type());
```

Invoking this static method tells the factory that any time it sees a base class, it needs to replace it with the specified child class. As a result, we can now rewrite the random test class:

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test);
    env env_h;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
    function build_phase(uvm_phase phase);
        //override type of object the factory creates for the base tester
        base_tester::type_id::set_type_override(random_tester::get_type());
        //instantiate the env class
        env_h = env::type_id::create("env_h",this);
    endfunction : build_phase

endclass
```

All tests will override the factory with the tester that they need. Meanwhile, the env class will handle setting up the environment and kicking off the objects. We have officially separated the testbench structure from the test class. All that needs to be done by the test class now is to plug in the correct stimulus or tester type.

10 The Observer Design Pattern

The observer design pattern operates similar to a social media website. Using it, an object can create data then share it to the "world". From there, the objects listening can do whatever they want with their copy of the data. We call these objects subscribers. The observed object does not care who is listening or what they do with their copy of the data; it's only job is to put the data out there. Luckily, the UVM provides us with two classes that help implement this design pattern.

10.1 UVM Analysis Port

The uvm_analysis_port class allows us to send data to subscribers. It requires a three step process:

1. Declare a variable for the analysis port and define the type of data that will flow through it
2. Instantiate the analysis port in the build phase
3. Insert data in the analysis port by invoking the write method

Invoking the write method will make the provided data available to all the port subscribers. This base class also contains a connect method for connecting subscribers to the analysis port. The single agrument for this method is an object of type analysis_export.

10.2 UVM Subscriber

The uvm_subscriber class is derived from the uvm component class and allows us to connect to an analysis port. The class contains a data member of type analysis_export and a pure virtual data handling method called write.

10.3 Putting It All Together

We first implement our subcribers. The uvm subscriber class is parameterized to specify the data type that it will handle. This data type needs to match the type of the argument passed to the write method.

```
class sub1 extends uvm_subscriber #(int);
    `uvm_component_utils(sub1);
    //inherits analysis_export member from parent class
    protected real count;
    protected real total;

    function new void (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function void write (int t);
```

```
        total = total + t;
        count++;
    endfunction : write
endclass
```

Next, we create our analysis port in the observed class. This follows the three step process given above.

```
class obs_obj extends uvm_component;
    `uvm_component_utils(obs_obj);
    uvm_analysis_port #(int) obj_ap; //Step 1
    int data;

    function new void (string name, uvm_component parent);
        super.new(name, parent);
        data = 0;
    endfunction : new

    function void build_phase(uvm_phase phase);
        //Step 2
        obj_ap = new("obj_ap",this); //analysis ports do not use the factory
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        repeat(20) begin
            ... do something
            data = data + 1;
            obj_ap.write(data); //Step 3. Broadcast data
        end
        phase.drop_objection(this);
    endtask : run_phase
endclass : obs_obj
```

By invoking the write method with our data, we have automatically passed a copy of this data to all subscribers of the object.

The final part is to tell the subscribers what objects they should be listening to. This is done via the connect phase method. The connect phase occurs after the build phase is complete. The build phase is execute in a hierarchical fashion. The UVM will invoke the top-level build phase method then continue to travel down each level of the testbench hierarchy and invoke the pertinent build phase methods. The connect phase runs opposite in a bottom to top fashion. The test class will contain the connect phase that connects the subscribers to the observed as such.

```
class test extends uvm_test;
    `uvm_component_utils(test);

    obs_obj obs_obj_h;
    sub1 sub1_h;
    sub2 sub2_h;

    ...

    function void connect_phase(uvm_phase phase);
        obs_obj_h.obj_ap.connect(sub1_h.analysis_export);
        obs_obj_h.obj_ap.connect(sub2_h.analysis_export);
    endfunction
endclass
```

11 The Analysis Layer

All testbenches do two things: drive stimulus on a DUT, analyze what comes out. The first part is covered by our tester class. The coverage and scoreboard classes make up the second part in what is called the analysis layer in the UVM. In both the scoreboard and the coverage class, we require knowledge about the what command was issued to the DUT. What is the best way to disseminate this data? Well, we didn't just talk about analysis ports for nothing.

Our testbench structure will look like this. The tester will drive the BFM with stimulus. The BFM will then recognize the command given and the results produced and place them each in their own monitor (analysis port). The monitors will then broadcast the data to their subscribers. In our example, the command monitor will have the coverage and scoreboard objects as subscribes, while the results monitor will only have the scoreboard object as a subscriber.

11.1 Bridging the BFM and the Monitors

The BFM should be seen as the bridge between the testbench environment and the DUT. As a result, we want to be able to pass data into our monitors from within the BFM. The first requirement to do this is to create monitor handles within our BFM:

```
interface tinyalu_bfm;
    import tinyalu_pkg::*; //contains monitor definitions
    cmd_mon cmd_mon_h;
    rslt_mon rslt_mon_h;
...
```

The BFM is now ready to store our monitor objects. We will create these objects in the build phase of the respective monitor classes. The created objects will then have their handles passed to the BFM, so we are referencing the correct monitors.

```
class cmd_mon extends uvm_component;
    `uvm_component_utils(cmd_mon);
    uvm_analysis_port #(command_s) cmd_ap;

    function void build_phase(uvm_phase phase);
        virtual interface tinyalu_bfm bfm;
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
        bfm.cmd_mon_h = this; //connect monitor to bfm
        cmd_ap = new("cmd_ap",this); //inst analysis port
    endfunction
endclass : cmd_mon
```

Now, the BFM can pass data into the testbench through the monitor.

11.2 From DUT to BFM to Monitor

Always blocks are implemented in the BFM to send the data that the monitor wants. Here we are sending the command data to the command monitor once we see a new command has been issued.

```
    always @(posedge clk) begin : cmd_mon
        bit new_cmd;
        if(!start)
            new_cmd = 1'b1;
        else
            if(new_cmd) begin
                cmd_mon_h.write_to_monitor(A, B, op);
                new_cmd = (op == 3'b0); //nop
            end
    end : cmd_mon
```

The monitor class takes the command passed by the BFM and broadcasts it to the subscribers.

```
typedef struct{
    byte A;
    byte B;
    logic [2:0] op;
} cmd_s;

class cmd_mon extends uvm_component;
    `uvm_component_utils(cmd_mon);
    uvm_analysis_port #(cmd_s) cmd_ap; //Step 1. Create handle

    function void build_phase (uvm_phase phase);
        //get bfm handle
        cmd_ap = new("cmd_ap", this); //Step 2. Instantiate
    endfunction : build_phase

    function void write_to_monitor(byte A, byte B, logic [2:0] op)
        cmd_s cmd;
        cmd.A = A;
        cmd.B = B;
        cmd.op = op;
        cmd_ap.write(cmd_s); //Step 3. Broadcast
    endfunction : write_to_monitor
endclass : cmd_mon
```

What we have put together so far is the BFM will monitor for a new command. Upon seeing a new command, it will transmit the command data to the command monitor. The command monitor will package the data in a struct then place that struct on the analysis port for the subscribers to see.

11.3 Subscribing to a Single Analysis Port

The coverage class can now be hooked up to subscribe to the command monitor analysis port. The actual hook up will occur in the test class as above.

```
class coverage extends uvm_subscriber #(cmd_s);
    `uvm_component_utils(coverage);
    byte A;
    byte B;
    logic [2:0] op;
    ...
    function new write (cmd_s t);
        A = t.A;
        B = t.B;
        op = t.op;
        //collect coverage stats
    endfunction : write
endclass : coverage
```

The write method must have one argument that is called t. The coverage class now receives a nice struct from the monitor and does not have to worry about decoding pin outputs.

11.4 Subscribing to Multiple Analysis Ports

The scoreboard class does not have as simple a use case as it requires subscriptions to both the command and result monitors. The simplest way to approach this is to instantiate a second subscriber object in the class and use that to subscribe to the second port. In short, when we connect the export to the port, we need to be able to specify which export we are referring to.

```
class scoreboard extends uvm_subscriber #(shortint);
    `uvm_component_utils(scoreboard);
    uvm_tlm_analysis_fifo #(cmd_s) cmd_f;

    function void build_phase (uvm_phase phase);
        cmd_f = new("cmd_f",this);
    endfunction : build_phase

    function void write (shortint t);
        cmd_s cmd;
        shortint pred_result;
        cmd.op = nop;
        do
            if(!cmd_f.try_get(cmd))
                $fatal();
        while((cmd.op = nop) || (cmd.op == rst_op));
        //process results
        case(cmd.op)
            add_op : pred_result = cmd.A + cmd.B;
            and_op : pred_result = cmd.A & cmd.b;
            xor_op : pred_result = cmd.A ^ cmd.B;
            mul_op : pred_result = cmd.A * cmd.b;
        endcase

        if(pred_result != t)
            $error();
    endfunction : write
endclass : scoreboard
```

The uvm tlm analysis fifo is a parameterized class that provides an analysis export and a try get pop method. In the above example, we expect the broadcast of a result to be accompanied by a command in our fifo. Upon receiving the result, we poll the fifo until we see a command that is not nop or reset. If we get a result without a command being present in the fifo, something went wrong. Upon seeing a valid command, we calculate the predicted result and display if there are discrepancies between the result we calculated and the one that was broadcast.

11.5 Subscribing to Monitors

The last step in our process is connecting the analysis ports to the analysis exports in the connect phase of our test class.

```
    function void connect_phase (uvm_phase phase);
        cmd_mon_h.cmd_ap.connect(coverage_h.analysis_export);
        rslt_mon_h.rslt_ap.connect(scoreboard_h.analysis_export);
        //Hook up second analysis export to fifo
        cmd_mon_h.cmd_ap.connect(scoreboard_h.cmd_f.analysis_export);
    endfunction
```

12 Interthread Communication

The previous section only contains intrathread communication methods. What happens when we want to communicate across threads? Object-oriented SystemVerilog provides interthread communication constructs such as semaphores and mailboxes. The UVM takes these and builds a universal solution that has two parts:

- Ports - Objects instantiated in uvm components that allow the run phase to talk with other threads. Put ports send data to other threads and get ports receive data from other threads.

- TLM FIFOs - Objects that connect a put port and a get port. 1 deep FIFOs. TLM stands for transaction-level-modeling, but the name is legacy.

12.1 The Put Port

The put port will be instantiated in our producer. The uvm_put_port class is a parameterizable class that stores one element of the given data type. The put method will place a data sample in a TLM FIFO. Once full, invoking the put method will pause execution of the thread until the consumer reads out of the FIFO.

```
class producer extends uvm_component;
    `uvm_component_utils(producer);
    protected int shared;
    uvm_put_port #(int) put_port_h;

    function void new (string name, uvm_component parent);
        super.new(name, parent);
        shared = 0;
    endfunction

    function void build_phase (uvm_phase phase);
        put_port_h = new("put_port_h",this);
    endfunction : build_phase

    task run_phase (uvm_phase phase);
        phase.raise_objection(this);
        repeat (3) begin
            put_port_h.put(shared++);
        end
        phase.drop_objection(this);
    endtask : run_phase
endclass : producer
```

12.2 The Get Port

The get port operates almost identically to the put port. It is a parameterized class that retrieves data from a TLM FIFO. If the get method is invoked while the FIFO is empty, it will pause the thread.

```
class consumer extends uvm_component;
    `uvm_component_utils(consumer);
    protected int shared;
    uvm_get_port #(int) get_port_h;

    function void new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void build_phase (uvm_phase phase);
        get_port_h = new("get_port_h",this);
    endfunction : build_phase

    task run_phase (uvm_phase phase);
        phase.raise_objection(this);
        repeat (3) begin
            get_port_h.get(shared);
        end
        phase.drop_objection(this);
    endtask : run_phase
endclass : consumer
```

12.3 Connecting the Ports

The producer, consumer and TLM FIFO are instantiated and connected in a uvm test class. The TLM FIFO should be parameterized with the same data type as the producer and consumer. The test class build phase handles the instantiation, and the connect phase connects the ports through the FIFO. The put and get ports both contain connect methods. The TLM FIFO provides put_export and get_export objects for connecting, similar to the uvm subscriber.

```
class test extends uvm_test;
    `uvm_component_utils(test);
    producer producer_h;
    consumer consumer_h;
    uvm_tlm_fifo #(int) fifo_h

    function build_phase(uvm_phase phase);
        producer_h = new("producer_h",this);
        consumer_h = new("consumer_h",this);
        fifo_h = new("fifo_h",this);
    endfunction

    function connect_phase(uvm_phase phase);
        producer_h.put_port_h.connect(fifo_h.put_export);
        consumer_h.get_port_h.connect(fifo_h.get_export);
    endfunction
endclass : test
```

12.4 Non-blocking Communication

The blocking style of the put and get methods above may not always be wanted, especially if we are employing other blocking elements such as clock triggers. Instead, we can use the try_get and try_put methods. These methods will attempt to pop or push the FIFO and return a 1 if successful, 0 if unsuccessful. After attempting a pop or push, the thread execution will continue regardless of the outcome.

```
task run_phase (uvm_phase phase);
    super.run_phase(phase);
    forever begin
        @(posedge clk_bfm.clk);
        if(get_port_h.try_get(shared))
            $display("Success");
    end
endtask : run_phase
```

13 Integrating Put and Get Ports in a Testbench

The theme of the UVM is to take classes that perform several functions and break them down into single function classes that are easier to debug and reuse. However, we have been violating this key rule in our tester class. Our tester class so far has been responsible for choosing the operation type and generating stimulus on the BFM. In this section, we use the put and get ports to separate the tester class into the tester class and the driver class. The tester class will handle choosing the operation type, and the driver class will handle driving this operation on the DUT through the BFM. These two objects communicate to each other through a FIFO.

13.1 The Big Picture So Far

I'm including a diagram of what our testbench looks like now that we've introduced all these other pieces.

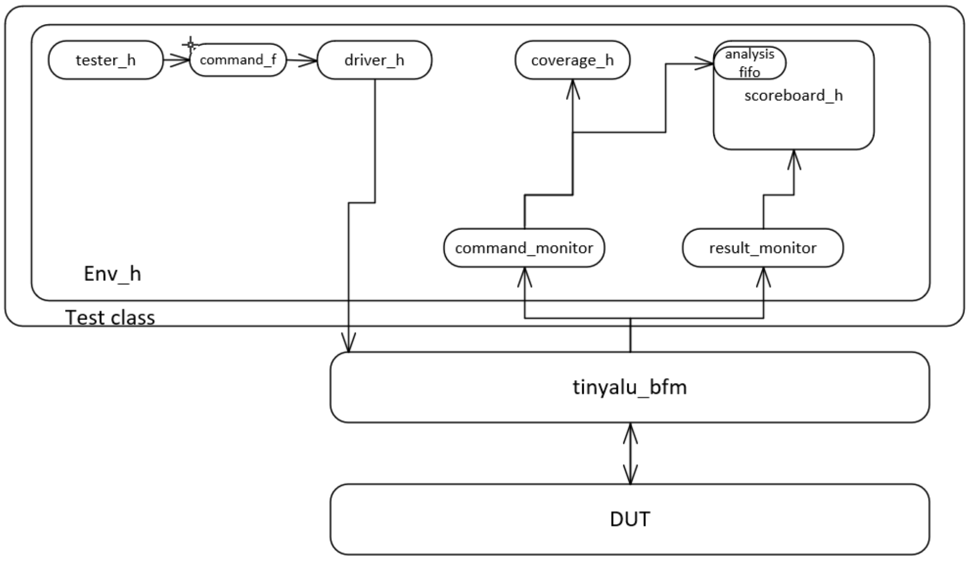

Figure 3: Top-level Testbench

The tester and driver classes are connected via a TLM FIFO. This connection occurs in the environment class.

```
class env extends uvm_env;
    `uvm_component_utils(env);
    tester tester_h;
    driver driver_h;
    uvm_tlm_fifo #(cmd_s) command_f;
    ...
    function void connect_phase (uvm_phase phase);
        tester_h.put_port_h.connect(command_f.put_export);
        driver_h.get_port_h.connect(command_f.get_export);
        ...
    endfunction
endclass : env
```

13.2 The Division of the Tester and Driver

Our base tester class will now be removed from driving the BFM directly. Instead, all the tester class needs to do is put the data into the TLM FIFO.

```
virtual class base_tester extends uvm_component;
    `uvm_component_utils(base_tester);
    uvm_put_port #(cmd_s) put_port_h;

    function void build_phase(uvm_phase phase);
        put_port_h = new("put_port_h",this);
    endfunction : build_phase

    pure virtual function operation_t get_op();

    task run_phase(uvm_phase phase);
```

```
        byte unsigned iA;
        byte unsigned iB;
        operation_t   op_set;
        cmd_s  command;

        phase.raise_objection(this);
        command.op = rst.op;
        command_port.put(command); //place command in FIFO
        repeat (1000) begin
            command.op = get_op();
            command.A = get_data();
            command.B = get_data();
            command_port.put(command);
        end
        #500;
        phase.drop_objection(this);
    endtask : run_phase
endclass : base_tester
```

The driver will then take the data out of the TLM FIFO and interface with the BFM to get that operation data into the DUT. In this case, we use the BFM's send op method.

```
class driver extends uvm_component;
    `uvm_component_utils(driver);
    virtual interface tinyalu_bfm bfm;
    uvm_get_port #(cmd_s) command_port;

    function void build_phase(uvm_phase phase);
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
        command_port = new("command_port",this);
    endfunction

    task run_phase(uvm_phase phase);
        cmd_s cmd;
        shortint result;
        forever begin
            command_port.get(cmd);
            bfm.send_op(cmd.A, cmd.B, cmd.op, result)
        end
    endtask : run_phase
endclass : driver
```

As we can see above, the driver will wait patiently until the tester produces a new operation that needs to be sent.

14 Reporting

The UVM provides systematic ways to report events through reporting macros. There are four reporting macros that display messages of increasing severity:

- `uvm_info(Message ID String, Message String, Verbosity)
- `uvm_warning(Message ID String, Message String)
- `uvm_error(Message ID String, Message String)
- `uvm_fatal(Message ID String, Message String)

The message ID string identifies the type of message, and the message string is what is printed to the terminal. All reporting macros will output the following elements: severity, call file and line number, time of error, call's location in UVM hierarchy, message ID string, and message string.

14.1 Verbosity Levels

The UVM limits what messages will print through the use of verbosity levels. The verbosity level of a simulation will set the highest priority verbosity that will print. The UVM ships with six verbosity levels:

- UVM_NONE
- UVM_LOW
- UVM_MEDIUM
- UVM_HIGH
- UVM_FULL
- UVM_DEBUG

NONE is the lowest level and DEBUG is the highest. These are defined in an enum in the UVM. If the verbosity level is set at UVM_MEDIUM, then all HIGH, FULL, and DEBUG level messages will not print.

14.2 Setting the Verbosity Level

The verbosity level can be set at a global level using the UVM_VERBOSITY plusarg:

```
xrun +UVM_VERBOSITY=UVM_HIGH
```

The verbosity level can also be set in the UVM hierarchy to define verbosity at a more granular level. The UVM hierarchy is defined after the build phase when all the classes are instantiated. As a result, we need a phase that lives just before the run phase. This phase is called the end of elaboration phase. Every uvm component contains a method to set the verbosity level at that point.

```
function void end_of_elaboration_phase(uvm_phase phase);
    //Set verbosity level in only scoreboard
    scoreboard_h.set_report_verbosity_level(UVM_HIGH);
    //Set verbosity level in scoreboard and all objects below scoreboard
    scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

14.3 Disabling Messages

UVM macros are able to perform a number of different actions. These are defined in the UVM action type enum and include:

- UVM_NO_ACTION
- UVM_DISPLAY
- UVM_LOG
- UVM_COUNT
- UVM_EXIT
- UVM_CALL_HOOK
- UVM_STOP

These actions can be added as an argument to the verbosity level setting:

```
function void end_of_elaboration_phase(uvm_phase phase);
    scoreboard_h.set_report_verbosity_level(UVM_HIGH, UVM_NO_ACTION);
    //Actions can be OR'd together
    scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH, UVM_NO_ACTION|UVM_LOG);
endfunction : end_of_elaboration_phase
```

15 Class Hierarchies and Deep Operations

To avoid copying code in extended classes, we can use deep operations. For the case of methods, we call the super.method() to carry on the parent class functionality without having to worry about what it does. Pretend we have two classes with a convert2string method:

```
class parent;
    string name;
    function string convert2string();
        return name;
    endfunction : convert2string
endclass : parent

class child extends parent;
    string name;
    function string convert2string();
        //Deep operation
        return {super.convert2string(), name};
    endfunction : convert2string
endclass : child
```

The benefit of this approach is that we can change the convert2string function in the parent class, and that change will automatically show up in the extended classes. Furthermore, we can even add a class between parent and child while preserving the intended operation.

```
class parent;
    string name;
    function string convert2string();
        return name;
    endfunction : convert2string
endclass : parent

class interm extends parent;
    string middle_name;
    function string convert2string();
        return {super.convert2string(), middle_name};
    endfunction : convert2string
endclass : interm

class child extends interm;
    string name;
    function string convert2string();
        return {super.convert2string(), name};
    endfunction : convert2string
endclass : child
```

15.1 Deep Copying

Another place where deep operations make life easier is with copying objects. A common pitfall is doing the following to copy an object:

```
obj_2_h = obj_1_h;
```

All that we've done above is stored the handle for object 1 in a different handle. As a result, the object 1 and object 2 handles are referring to the same object, which is not what we wanted.

To solve this, we need to invoke a do copy method with the object we want to copy onto.

```
obj_2_h = new();
obj_2_h.do_copy(obj_1_h);
```

When defining this do copy method, we want to include all data members that the object being copied has inherited. This requires that we include a super.do_copy in all of our do copy methods up the class hierarchy, meaning all do copy methods will need to take the same input type. However, the child class do copy will want a child object input while the parent class will want a parent object input. This is an issue.

The solution to this is similar to quickly climbing all the way up the class hierarchy and then walking back down. In the do copy method of each extended class, we take in an object that is the same type as the base object. We then call the do copy of the parent class with this object, cast the object on to the current class type, and add our class specific members back. Naturally, let's write out an example using lions.

```
class lion; //base class
    int age;

    function void do_copy(lion copied_lion);
        this.age = copied_lion.age;
    endfunction : do_copy
endclass : lion

class captive_lion extends lion;
    string name;

    function void do_copy(lion copied_lion);
        captive_lion copied_captive_lion;
        super.do_copy(copied_lion);
        $cast(copied_captive_lion, copied_lion);
        this.name = copied_captive_lion.name;
    endfunction : do_copy
endclass : captive_lion

class circus_lion extends captive_lion;
    int num_tricks;

    function void do_copy(lion copied_lion);
        circus_lion copied_circus_lion;
        super.do_copy(copied_lion);
        $cast(copied_circus_lion, copied_lion);
        this.num_tricks = copied_circus_lion.num_tricks;
    endfunction : do_copy
endclass : circus_lion
```

15.2 A Note on Polymorphism

I want to make a remark on an aspect of polymorphism that I have found to be insufficiently covered in texts on OOP SystemVerilog. Pretend that we have a child class extended from a parent class. The child class inherits a variable called var1 from parent. We know that we are able to upcast the child class to a parent class handle thanks to polymorphism. Now, what happens if the child class has added a variable called var2. Can this variable be accessed as such:

```
parent parent_h;
```

```
child child_h;
initial begin
    child_h = new();
    parent_h = child_h;
    $display(parent_h.var2);
end
```

The answer is no. We would get an error because the parent class does not contain a member called var2. However, we can still access var2 by downcasting back onto a child object.

```
initial begin
    child_h = new();
    parent_h = child_h; //Upcast
    $cast(child_h,parent_h); //Downcast
    $display(child_h.var2);
end
```

It is a misconception to think that storing a child object in a parent handle has destroyed var2. When we instantiated the child object, we allocated memory for all of it's members. By storing the child object in the parent handle, we are saying that we no longer consider the memory storing var2 to be a part of the object. This does not mean the memory disappears. Instead, we are acting on a subset of the total object memory. Upon casting the object back to a child, we can now act on the full memory allocated.

This leads to an interesting corollary: we can only downcast the parent object onto an object of the same type as the underlying object. The following code is valid:

```
parent parent_h;
child child_h;
child child2_h;
initial begin
    child_h = new();
    parent_h = child_h;
    $cast(child2_h,parent_h);
end
```

16 UVM Transactions

The motto that has guided us so far is breaking the testbench into smaller pieces such that each piece does not need to know the inner-workings of another piece to function. However, we have not done this with our data yet. All we've been doing is passing in a struct and getting a int output. Instead, we can transform the data into a class so that it has methods to operate on the data. Therefore, we are also able to pass class handles around the testbench. Classes that store data are called transactions.

Transactions contain both the data and methods to operate on the data including:

- convert2string - Provide string with data values in it
- do_copy - Copy transaction of same class onto this transaction
- do_compare - Compare transaction against transaction of same class
- Randomize data fields using SV's randomize methods with constraints

Transactions are extended from the uvm_transaction base class and should override the do copy, do compare, and convert2string methods.

16.1 Random Data Fields

Note: This section will extensively utilize the randomization features of the SystemVerilog language. These features are covered in-depth in SystemVerilog for Verification by Spear and Tumbush.
All SV classes provide an implicit method called randomize, which will assign random values to its

data fields that have the keyword rand. Constraints can be placed on the random fields to enforce values and probability distributions.

```
class command_transaction extends uvm_transaction;
    `uvm_object_utils(command_transaction);
    rand byte unsigned A;
    rand byte unsigned B;
    rand operation_t op;
    //Equal chance of all 0's, all 1's, or random
    constraint data {
    A dist {8'h00:=1, [8'h01:8'hFE]:=1, 8'hFF:=1};
    B dist {8'h00:=1, [8'h01:8'hFE]:=1, 8'hFF:=1};}

    function new (string name = "");
        super.new(name);
    endfunction : new
endclass : command_transaction
```

The UVM transaction class extends the UVM object class. As a result, it exists outside of the UVM hierarchy and does not require a parent to be specified in its constructor.

16.2 Copies, Clones, Compares, and Converts

The UVM object base class provides virtual copy and clone methods. The copy method is used to copy all of an object's data onto another object of the same type. Because we are inheriting this method, we need to follow the framework that the UVM object class has set. This just means that the argument to do copy has to be a UVM object with the name rhs (right hand side).

```
function void do_copy(uvm_object rhs);
    command_transaction copied_transaction_h;

    if(rhs == null)
        `uvm_fatal("CMD TRANS","Tried to copy from null param);
    if(!$cast(copied_transaction_h,rhs))
        `uvm_fatal("CMD TRANS","Tried to copy wrong type);

    super.do_copy(rhs); //deep copy parent data
    A = copied_transaction_h.A;
    B = copied_transaction_h.B;
    op = copied_transaction_h.op;
endfunction : do_copy
```

We included two type checks to make sure the passed handle is not null and references a valid type. This method will be invoked as obj2.do_copy(obj1);

Using objects for storing data allows us to share the object handle around the testbench instead of creating separate instances. This is all fine and dandy, but requires a design rule called Mandatory Obligatory Object Copy On Write (MOOCOW). This rule essentially says that you can use the object handle all you want, but before you change the data contained in the object you must make a copy and alter that copy instead. This ensures that you aren't screwing over all the other users of the transaction. The UVM supports MOOCOW through the do clone method. The do clone method uses the do copy method to create a UVM object. This object then needs to be casted onto the class type.

```
function command_transaction do_clone();
    command_transaction clone;
    uvm_object tmp;

    this.do_copy(tmp);
    $cast(clone, tmp);
    return clone;
endfunction : do_clone
```

Our final method is the do compare method used to check that two objects are the same type and have the same data member values. Do compare performs a deep operation.

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    command_transaction compared_transaction_h;
    bit same;
    if(rhs == null)
        `uvm_fatal();

    if($cast(compared_transaction_h, rhs))
        same = 0;
    else
        same = super.do_compare(rhs, comparer) &&
            (compared_transaction_h.A == A) &&
            (compared_transaction_h.B == B) &&
            (compared_transaction_h.op == op);
    return same;
endfunction : do_compare
```

We once again type check on the passed in object then we perform value checks on the data members.

The final method is convert2string. This is nifty for displaying the object values.

```
function string convert2string();
    string s;
    s = $sformatf("A: %2h B: %2h op: %s", A, B, op.name());
    return s;
endfunction : convert2string
```

16.3 Integrating Transactions

Here are the steps we will take to integrate transactions into our testbench:

1. Create a result_transaction class to hold results
2. Create an add_transaction class that extends command transaction to generate only add operations
3. Rename base_tester to tester
4. Modify command monitor to create a command_transaction
5. Modify result monitor to create a result_transaction
6. Modify the scoreboard to use reseult_transaction's compare method
7. Modify the add_test to use the add_transaction

Steps 1 and 2. Step 1 will define the result transaction class along with all the methods we just mentioned. Step 2 will extend the command transaction as a add transaction class, which just enforces that the op be an add operation. This method allows us to use the same tester object and swap out transaction types instead.

```
//Step 1
class result_transaction extends uvm_transaction;
    `uvm_object_utils(result_transaction);
    ...

//Step 2
class add_transaction extends command_transaction;
    `uvm_object_utils(add_transaction);
    constraint add_only {op == add_op};
    function new (string name ="");
        super.new(name);
    endfunction
endclass : add_transaction
```

Step 3. The base tester class will now be the only tester class. We generate our command transaction in the tester using the factory, which allows us to override the transaction type. The class objects are then randomize according to the constraints of the class. This means that we can use the factory override to get only add operations.

```
    command = command_transaction::type_id::create("command");
    repeat(10) begin
        assert(command.randomize);
        command_port.put(command);
    end
```

Step 4. The command monitor will now take data from the BFM and package it into a transaction instead of a struct. This code in the command monitor changes minimally.

```
function void write_to_monitor(byte A, byte B, operation_t op);
    command_transaction cmd;
    cmd = new("cmd");
    cmd.A = A;
    cmd.B = B;
    cmd.op = op;
    ap.write(cmd);
endfunction : write_to_monitor
```

Step 5. A similar story with the result monitor.

```
function void write_to_monitor(shortint r);
    result_transaction result;
    result = new("result");
    result.r = r;
    ap.write(result);
endfunction : write_to_monitor
```

Step 6. The scoreboard is reworked to use transactions instead of structs. We are now passed a result transaction called t and a command transaction. We turn the command transaction into a predicted results transaction and compare it to the passed result transaction.

```
        predicted = predict_results(cmd);
        if(!predicted.compare(t))
            $fatal();
        else
            pass;
    endfunction : write
endclass : scoreboard
```

Step 7. The factory transaction type can now be overridden at the UVM test level to create a test that targets only add operations without creating a new tester object.

```
class add_test extends uvm_test;
    `uvm_component_utils(add_test);

    function void build_phase(uvm_phase phase);
        tinyalu_transaction::type_id::set_type_override(add_transaction::get_type());
        super.build_phase(phase);
    endfunction : build_phase
```

17 Not So Secret Agents

The wonder of modularity is made possible by defining strict and constant interfaces. If we know how to connect to the object, we can ignore how it works at a deeper level. The UVM supports modularity through agents, configuration objects, and the configuration database.

17.1 The UVM Agent

Figure 4 is a schematic of our current testbench setup.

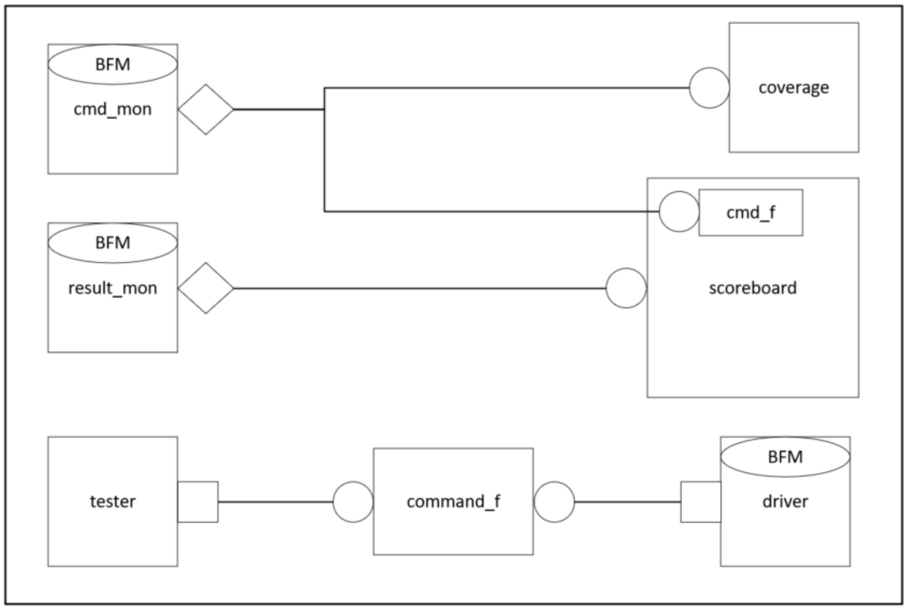

Figure 4: Diagram of Current Testbench

A major limitation to this setup is its scalability. If we added an additional tinyALU to the design, we would have to copy and paste the all of these objects to reuse them. The natural step forward is to encapsulate the testbench components into what is call an agent. Figure 5 visually represents what our agent encapsulates.

All components of the testbench are contained within the agent. The command and result analysis ports are outputs so that other higher level components may monitor what is happening in our agent. The dashed line around the tester and driver indicate that these objects are only instantiated if the user plans to drive the BFM using this agent. Without them, the user can still monitor the DUT using the agent. The agent abstracts away all the inner-workings of our environment. All we have to connect to is the BFM, and analysis ports.

17.2 The Meat and Potatoes

The agent class will essentially replace the env class that we talked about in previous sections. An agent class will have the additional functionality of a configuration class to transfer the BFM and control whether the agent creates stimulus.

```
class tinyalu_agent_config;
    virtual interface tinyalu_bfm bfm;
    protected uvm_active_passive_enum is_active;

    function new(virtual interface tinyalu_bfm bfm,uvm_active_passive_enum is_active);
        this.bfm = bfm;
        this.is_active = is_active;
    endfunction : new
```

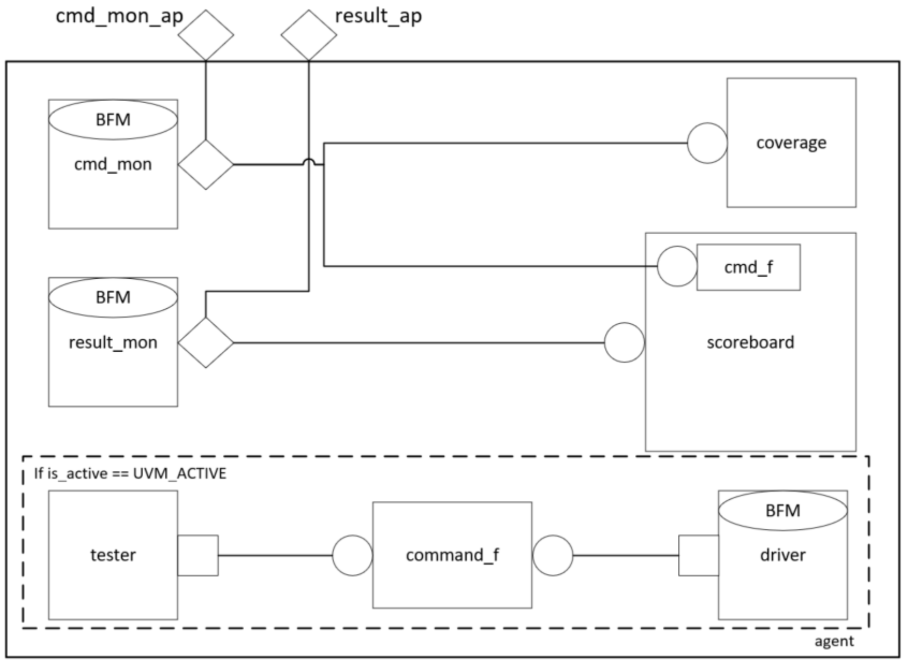

Figure 5: UVM Agent

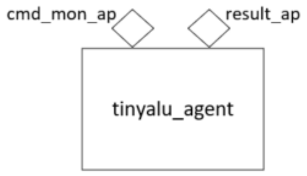

Figure 6: UVM Agent Interface

```
    function uvm_active_passive_enum get_is_active();
        return is_active;
    endfunction : get_is_active
endclass : tinyalu_agent_config

class tinyalu_agent extends uvm_agent;
    `uvm_component_utils(tinyalu_agent);
    tinyalu_agent_config tinyalu_agent_config_h;

    tester tester_h;
    driver driver_h;
    scoreboard scoreboard_h;
    coverage coverage_h;
    cmd_monitor cmd_monitor_h;
    rslt_monitor rslt_monitor_h;

    uvm_tlm_fifo #(cmd_transaction) command_f;
    uvm_analysis_port #(cmd_transaction) cmd_mon_ap;
    uuvm_analysis_port #(rslt_transaction) result_ap;

    function void build_phase(uvm_phase phase);
        if(!uvm_config_db#(tinyalu_agent_config)
        ::get(this,"","config",tinyalu_agent_config_h))
            $fatal();
        is_active = tinyalu_agent_config_h.get_is_active();

        if(get_is_active() == UVM_ACTIVE) begin : make_stim
            command_f = new("command_f",this);
            tester_h = tester::type_id::create("tester_h",this);
            driver_h = driver::type_id::create("driver_h",this);
        end

        cmd_monitor_h = cmd_monitor::type_id::create("cmd_monitor_h",this);
        rslt_monitor_h = rslt_monitor::type_id::create("rslt_monitor_h",this);
        coverage_h = coverage::type_id::create("coverage_h",this);
        scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);

        cmd_mon_ap = new("cmd_mon_ap",this);
        rslt_ap = new("rslt_ap",this);

    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        if(get_is_active() == UVM_ACTIVE) begin
            driver_h.command_port.connect(command_f.get_export);
            tester_h.command_port.connect(command_f.put_export);
        end

        //Analysis ports to top level
        cmd_monitor_h.ap.connect(cmd_mon_ap);
        rslt_monitor_h.ap.connect(rslt_mon_ap);

        cmd_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
        cmd_monitor_h.ap.connect(coverage_h.analysis_export);
        rslt_monitor_h.ap.connect(scoreboard_h.analysis_export);
    endfunction : connect_phase
```

This is a lot of code but it simply implements what we've been talking about. The uvm agent base class gives a is_active member of type uvm active passive enum, which contains UVM_ACTIVE and UVM_PASSIVE. This is used in the build phase to instantiate the tester, driver, and command fifo.

The handle for the agent configuration class is retrieved from the configuration database using uvm config db. This method is actually normally used for this purpose and not passing around BFMs.

17.3 A Note On Interfaces

The configuration objects described can function as the interface between objects in the UVM. For agents, we use configuration objects to pass an agent the value of is active and the relevant BFM handle. For environment classes, we use configuration objects to pass the various BFMs into the environment. In both of these cases, we are using the configuration class to transform these bits of information we want to send to an object into a standard form.

17.4 The Top Level

The top-level module is changed to work with uvm agents.

```
module top;
    import uvm_pkg::*;
    import tinyalu_pkg::*;
    `include "uvm_macros.svh"
    `include "tinyalu_macros.svh"
    tinyalu_bfm bfm1();
    tinyalu DUT1(bfm1.dest());

    tinyalu_bfm bfm2();
    tinyalu DUT2(bfm2.dest());

    initial begin
        uvm_config_db#(virtual interface bfm)::set(null,"*","bfm1",bfm1);
        uvm_config_db#(virtual interface bfm)::set(null,"*","bfm2",bfm2);
        run_test("dual_test");
    end
endmodule : top
```

The top level will instantiate the BFMs and DUTs, connect them, then place them into the configuration database where they can be retrieved. The user then calls run test with the test class name and the simulation environment handles the rest.

The dual test class takes the BFMs we stored and turns them into configuration objects for the environment class. The dual test class only instantiates an env object.

```
class env_config;
    virtual tinyalu_bfm bfm1;
    virtual tinyalu_bfm bfm2;

    function new (virtual tinyalu_bfm bfm1, bfm2);
        this.bfm1 = bfm1;
        this.bfm2 = bfm2;
    endfunction : new
endclass : env_config

class dual_test extends uvm_test;
    ...
    function void build_phase(uvm_phase phase);
        virtual tinyalu_bfm bfm1, bfm2;
        env_config env_config_h;
        if(!uvm_config_db#(virtual tinyalu_bfm)::get(this,"","bfm1",bfm1))
            `uvm_fatal();
        if(!uvm_config_db#(virtual tinyalu_bfm)::get(this,"","bfm2",bfm2))
            `uvm_fatal();

        env_config_h = new(bfm1, bfm2);
        uvm_config_db#(env_config)::set(this,"env_h*","config",env_config_h);
        env_h = env::type_id::create("env_h",this);
    endfunction : build_phase
endclass : dual_test
```

Note that the dual test class only contains a build phase method. Additionally, we are now using the uvm config db set method within the UVM hierarchy, which allows us to specify who can see the data we've set (more on this later).

17.5 The Environment

The environment class no longer serves as simply the location to instantiate all our testbench components. We now have multiple agents acting within our single environment class. As a result, the env class' job is to create configuration objects for these agents and store them in a place where they can be found. The uvm config db allows us to filter what type of objects have access to what type of data. Think of it as adding a specifier to the data. The env class looks like this:

```
if(!uvm_config_db#(env_config)::get(this,"","config",env_config_h))
```

```
    `uvm_fatal();
config1_h = new(.bfm(env_config_h.bfm1),.is_active(UVM_ACTIVE));
config2_h = new(.bfm(env_config_h.bfm2),.is_active(UVM_PASSIVE));

uvm_config_db#(tinyalu_agent_config)::set(this,"tinyalu_agent1_h*","config",config1_h);
uvm_config_db#(tinyalu_agent_config)::set(this,"tinyalu_agent2_h*","config",config2_h);

tinyalu_agent1_h = new("tinyalu_agent1_h",this);
tinyalu_agent2_h = new("tinyalu_agent2_h",this);
```

In this class, we retrieve the env configuration object from the test class. We then create agent configuration objects and store those in the configuration database. Note that our uvm config db set method is now being passed the "tinyalu_agent1_h" argument, which allows us to say that the string "config" for an object of this type will correspond to config1_h. Similarly, the string "config" to an object of type tinyalu_agent2_h will correspond to config2_h. Finally, the different agents are instantiated.

18 Sequences

The last section focused on introducing UVM agents to make the structure of the testbench modular. The section before that focused on UVM transactions to make data easy to create, compare, and transport by separating it from the structure. The final section in these notes will focus on what connects data and structure: test stimulus. We successfully pulled data out of structure, but now we must pull data stimulus out of structure.

The lingering issue is the presence of the tester class. Currently, the tester creates the set of transactions and feeds them into the testbench. That's two things, and we never want a class to do more than one thing. In a less convoluted way, we are able to override the transaction type to control data randomization, but we have to override the entire tester class to change how many and the order of transactions sent. In total, the goal of this section is to separate test stimulus from structure. This allows us to combine test stimulus if needed.

18.1 But What is a Sequence

We can convert the transaction-level testbench to use sequences through six steps:

1. Create a sequence item to carry our data
2. Replace tester with uvm_sequencer
3. Upgrade driver to support sequences
4. Instantiate and connect driver and sequencer in environment
5. Write UVM sequences
6. Write a test that starts the sequence using the sequencer

Step 1. The UVM sequence item is extended from the UVM transaction class. A sequence item will carry data from a sequence through the sequencer to a UVM driver.

```
class sequence_item extends uvm_sequence_item;
    `uvm_object_utils(sequence_item);
    function new (string name = "");
        super.new(name);
    endfunction : new

    rand byte A;
    rand byte B;
    rand operation_t op;
    shortint unsigned result;
    constraint
    ...
```

This looks exactly like our command transaction except it is an object of type uvm sequence item and has a result member.

Step 2. The sequencer class is responsible for taking in sequence items and transmitting them to the driver. The order of these sequences items is decided by a higher level sequence/the uvm test.

```
`include "sequence_item.svh"
typedef uvm_sequencer #(sequence_item) sequencer;
sequencer sequencer_h;
```

The type for the sequencer class should be defined in a DUT pkg file. The above is to simplify the process of instantiation through saying that we will always need a uvm sequencer class that accepts our sequence item definition.

Step 3. The UVM provides us with a uvm driver class that extends uvm component and is designed to interact with a uvm sequencer.

```
//Parameterized to work with sequence_item class
class driver extends uvm_driver#(sequence_item);
    `uvm_component_utils(driver);
    virtual tinyaly_bfm bfm;
    ...
    task run_phase(uvm_phase phase);
        sequence_item cmd;
        forever begin
            shortint unsigned result;
            seq_item_port.get_next_item(cmd);
            bfm.send_op(cmd.A, cmd.B, cmd.op, result);
            cmd.result = result;
            seq_item_port.item_done();
        end
    endtask : run_phase
```

By extending the uvm driver class, we have inherited a seq_item_port object. This object contains the methods get next item, which stalls until a sequence item is provided by the sequencer, and item done, which tells the sequencer that we are ready to receive another sequence item. Note that we are also returning the DUT result back to the caller, so the sequence can use the output of the DUT.

Step 4. The driver and the sequencer do not need a FIFO to connect to each other. The seq item port in the driver can be connected to the sequencer seq item export.

```
sequencer sequencer_h;
driver driver_h;
...
function void build_phase(uvm_phase phase);
    sequencer_h = new("sequencer_h", this);
    driver_h = driver::type_id::create("driver_h",this);
...
function connect_phase(uvm_phase phase);
    driver_h.seq_item_port.connect(sequencer_h.seq_item_export);
...
```

Step 5. A uvm sequence class exists outside of the UVM hierarchy. It is fed in through the sequencer. By extending the uvm sequence, a sequence inherits three things to interface with a sequencer:

- m_sequencer - Holds handle to the sequencer that takes our sequence item.
- task body() - Method invoked when UVM starts sequence.
- start item/finish item- Method pair that controls the sequencer.

The sequence class is responsible for creating sequence items and sending them to the sequencer.

```
class fibonacci_seq extends uvm_sequence #(sequence_item);
    `uvm_object_utils(fibonacci_seq);
    function new (stirng name = "");
        super.new(name);
    endfunction : new

    task body();
        byte n_minus_2 = 0;
        byte n_minus_1 = 1;
        sequence_item command;
        command = sequence_item::type_id::create("command");

        //Reset DUT
        start_item(command); //Wait for sequencer to be free
        command.op = rst_op;
        finish_item(command); //Release hold on sequencer once driver is done

        for(int ff = 3; fF<=14; ff++) begin
            start_item(command);
            command.A = n_minus_2;
            command.B = n_minus_1;
            command.op = add_op;
            finish_item(command);
            n_minus_2 = n_minus_1;
            n_minus_1 = command.result;
        end
    endtask : body
endclass : fibonacci_seq
```

The driver wrote the DUT result to our command handle, so we are able to access it after we give up control of the sequencer and use it.

Step 6. All tests will share the same build and end of elaboration phase methods. So we can define a base test class.

```
virtual class tinyalu_base_test extends uvm_test;
    env env_h;
    sequencer sequencer_h;

    function void build_phase (uvm_phase phase);
        env_h = env::type_id::create("env_h",this);
    endfunction : build_phase

    function void end_of_elaboration_phase(uvm_phase phase);
        sequencer_h = env_h.sequencer_h;
    endfunction : end_of_elaboration_phase

    function new (string name. uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass
```

We ignore the UVM agent here as its not the point of the chapter. The end of elaboration phase copies the sequencer handle from the environment so any extended test will have the handle to the sequencer.

18.2 Writing a Real Test

```
class fibonacci_test extends tinyalu_base_test;
    `uvm_component_utils(fibonacci_test);

    task run_phase(uvm_phase phase);
        fibonacci_sequence fibonacci;
        fibonacci = new("fibonacci");

        phase.raise_objection(this);
        fibonacci.start(sequencer_h);
        phase.drop_objection(this);
    endtask : run_phase
```

All sequences have a start method like above that takes in a sequencer handle as an argument then returns when the sequence is done. Now, we can use our test to simply pass our sequences a sequencer handler and watch the stimulus run. The test stimulus is officially separated from the structure.

18.3 Virtual Sequences

As shown above, we can launch any sequence in any order by invoking the start method with a sequencer handle. Virtual sequences allow us to circumvent passing in the sequencer handle. They're called virtual because they are called without a sequencer.

```
class runall_seq extends uvm_sequence #(uvm_sequence_item);
    `uvm_object_utils(runall_seq);

    protected reset_seq reset;
    protected maxmult_seq maxmult;
    protected random_seq random;
    protect sequencer sequencer_h;
    protected uvm_component uvm_component_h;

    function new(string name = "runall_seq");
        super.new(name);
        uvm_component_h = uvm_top.find("*.env_h.sequencer_h");
        if(uvm_component_h == null)
            `uvm_fatal();

        if(!$cast(sequencer_h, uvm_component_h))
            `uvm_fatal();

        reset = reset_seq::type_id::create("reset");
        maxmult = maxmult_seq::type_id::create("maxmult");
        random = random_seq::type_id::create("random");
    endfunction : new

    task body();
        reset.start(sequencer_h);
        maxmult.start(sequencer_h);
        random.start(sequencer_h);
    endtask : body
endclass : runall_seq
```

The uvm pkg provides the uvm top object of type uvm root. This objects includes a find method that can return a handle to an object within the UVM hierarchy by passing it the name of the object. In this case, we use a wildcard. The find method returns an object of type uvm component, so we need to downcast it as a sequencer. The sequence then uses the sequencer handle to launch the three other sequences in the body method.

18.4 Launching Virtual Sequences

```
class full_test extends tinyalu_base_test;
    `uvm_component_utils(full_test);
    runall_seq ra_seq;

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        ra_seq.start(null);
        phase.drop_objection(this);
    endtask : run_phase
    ...
```

The virtual sequence allows us to launch the sequence without passing it a sequencer handle.

We can extend this functionality by running sequences in parallel. The new method will store the sequencer handle in the sequence as m_sequencer. We use fork join to run the sequences in parallel. Once again, we are combining base sequences into larger sequences.

```
class parallel_test extends tinyalu_base_test;
    `uvm_component_utils(parallel_test);
    parallel_sequence parallel_h;

    function new(string name, uvm_component parent);
        super.new(name,parent);
        parallel_h = new("parallel");
    endfunction

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        parallel_h.start(null);
        phase.drop_objection(this);
    endtask : run_phase
    ...


class parallel_sequence extends uvm_sequence #(uvm_sequence_item);
    `uvm_object_utils(parallel_sequence);

    protected reset_seq reset;
    protected short_rand_seq short_rand;
    protected fibonacci_sequence fib_seq;

    function new(string name = "runall_seq");
        super.new(name);
        uvm_component_h = uvm_top.find("*.env_h.sequencer_h");
        if(uvm_component_h == null)
            `uvm_fatal();

        if(!$cast(m_sequencer, uvm_component_h))
            `uvm_fatal();

        reset = reset_seq::type_id::create("reset");
        maxmult = maxmult_seq::type_id::create("maxmult");
        random = random_seq::type_id::create("random");
    endfunction : new

    task body();
        reset.start(m_sequencer);
        fork
            fib_seq.start(m_sequencer);
            short_rand.start(m_sequencer);
        join
    endtask : body
endclass : parallel_sequence
```

The sequencer will arbitrate between the two sequences to ensure they both get fair access to the DUT. A variety of arbitration schemes are support, but the default is FIFO.

18.5 Wrap Up

The overall structure: a sequence item defines the test stimulus, a sequence contains sequence items and defines the order in which they are passed to the sequencer, the sequencer transmits the sequence items sent from the sequence to the driver, the driver uses the BFM to drive the stimulus on the DUT, the monitor uses the BFM to determine when the DUT has produced a result, the result is sent to the coverage collector and the scoreboard through analysis ports, the scoreboard receives the result and the stimulus to create a predicted result and check the two, the coverage confirms that we have hit all needed test points. The agent encapsulates our testbench structure/configuration and provides stimulus and result analysis ports. The environment encapsulates our agents and their configurations. The test will instantiate the environment and sequences needed, then connect up to the sequencer and the rest is handled by the UVM.

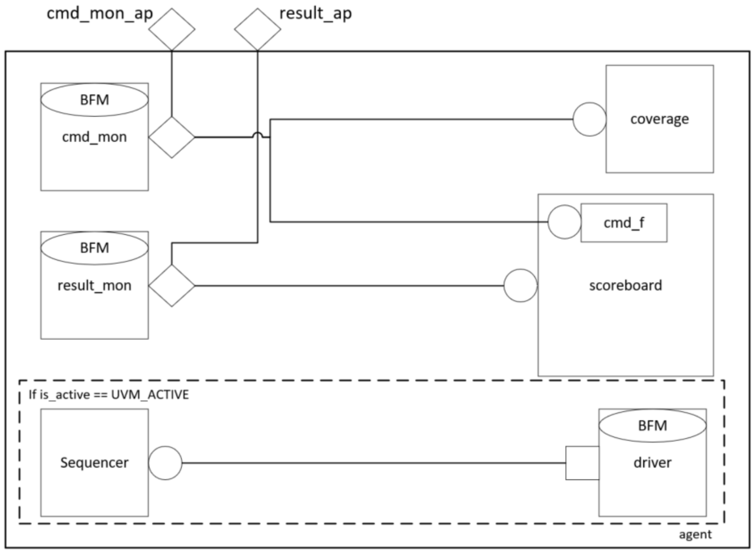

Figure 7: The Components of a UVM Testbench Environment

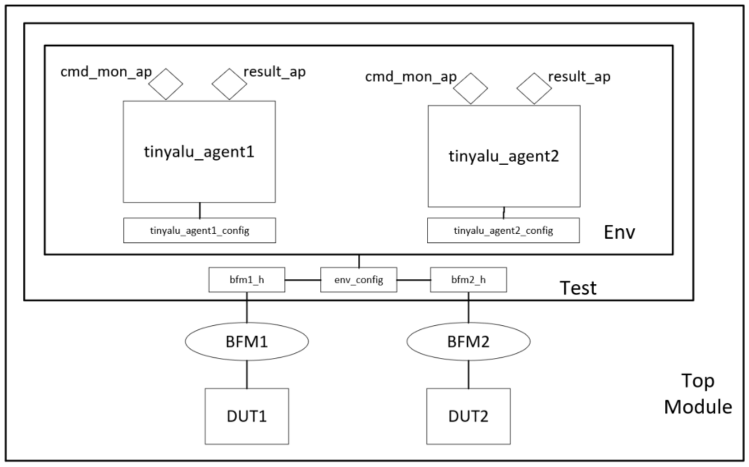

Figure 8: The Top Level View of a UVM Test